Towards Graph Programs for Graph Algorithms

Detlef Plump and Sandra Steinert

Department of Computer Science, The University of York
York YO10 5DD, UK
{det,sandra}@cs.york.ac.uk

Abstract. Graph programs as introduced by Habel and Plump [8] provide a simple yet computationally complete language for computing functions and relations on graphs. We extend this language such that numerical computations on labels can be conveniently expressed. Rather than resorting to some kind of attributed graph transformation, we introduce conditional rule schemata which are instantiated to (conditional) double-pushout rules over ordinary graphs. A guiding principle in our language extension is syntactic and semantic simplicity. As a case study for the use of extended graph programs, we present and analyse two versions of Dijkstra's shortest path algorithm. The first program consists of just three rule schemata and is easily proved to be correct but can be exponential in the number of rule applications. The second program is a refinement of the first which is essentially deterministic and uses at most a quadratic number of rule applications.

1 Introduction

The graph transformation language introduced by Habel and Plump in [8] and later simplified in [7] consists of just three programming constructs: nondeterministic application of a set of rules (in the double-pushout approach) either in one step or as long as possible, and sequential composition. The language has a simple formal semantics and is both computationally complete and minimal [7]. These properties are attractive for formal reasoning on programs, but the price for simplicity is a lack of programming comfort.

This paper is the first step in developing the language of [7] to a programming language GP (for *graph programs*) that is usable in practice. The goal is to design – and ultimately implement – a semantics-based language that allows high-level problem solving by graph transformation. We believe that such a language will be amenable to formal reasoning if programs can be mapped to a core language with a simple formal semantics. Also, graphs and graph transformations naturally lend themselves to visualisation which will facilitate the understanding of programs.

The language of [7] has no built-in data types so that, for example, numerical computations on labels must be encoded in a clumsy way. We therefore extend graph programs such that operations on labels are performed in a predefined algebra. Syntactically, programs are based on rule schemata labelled with terms over the algebra, which prior to their application are instantiated to ordinary

H. Ehrig et al. (Eds.): ICGT 2004, LNCS 3256, pp. 128–143, 2004.

double-pushout rules. In this way we can rely on the well-researched double-pushout approach to graph transformation [2, 6] and avoid resorting to some kind of attributed graph transformation. We also introduce conditional rule schemata which are rule schemata equipped with a Boolean term over operation symbols and a special `edge` predicate. This allows to control rule schema applications by comparing values of labels and checking the (non-)existence of edges.

To find out what constructs should be added to the language of [7] to make GP practical, we intend to carry out various case studies. Graph algorithms are a natural choice for the field of such a study because the problem domain need not be encoded and there exists a comprehensive literature on graph algorithms. In Section 7 we present and analyse two graph programs for Dijkstra's shortest path algorithm. The first program contains just three rule schemata but can be inefficient, while the second program is closer to Dijkstra's original algorithm and needs at most a quadratic number of rule applications. We prove the correctness of the first program and the quadratic complexity of the second program to demonstrate how one can formally reason on graph programs.

In general, we want to keep the syntax and semantics of GP as simple a possible while simultaneously providing sufficient programming comfort. Of course there is a trade-off between these aims; for example, we found it necessary to introduce a while loop in order to efficiently code Dijkstra's algorithm in the second program.

2 Preliminaries

A *signature* $\Sigma = (S, OP)$ consists of a set S of *sorts* and a family $OP = (OP_{\overline{s},s})_{\overline{s}\in S^*, s\in S}$ of *operation symbols*. A family $X = (X_s)_{s\in S}$ of *variables* consists of sets X_s that are pairwise disjoint and disjoint with OP. The sets $T_{OP,s}(X)$ of *terms* of sort s are defined by $x, c \in T_{OP,s}(X)$ for all $x \in X_s$ and all $c \in OP_{\lambda,s}$, and $op(t_1, \ldots, t_n) \in T_{OP,s}(X)$ for all $op \in OP_{s_1\ldots s_n,s}$ and all $t_1 \in T_{OP,s_1}(X), \ldots, t_n \in T_{OP,s_n}(X)$. The set of all terms over Σ and X is denoted by $T_\Sigma(X)$.

A *Σ-algebra* A consists of a family of nonempty sets $(A_s)_{s\in S}$, elements $c_A \in A_s$ for all $c \in OP_{\lambda,s}$, and functions $op_A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for all $op \in OP_{s_1\ldots s_n,s}$.

An *assignment* $\alpha\colon X \to A$ is a family of mappings $(\alpha_s\colon X_s \to A_s)_{s\in S}$. The extension $\hat{\alpha}\colon T_\Sigma(X) \to A$ of α is defined by $\hat{\alpha}(x) = \alpha(x)$ and $\hat{\alpha}(c) = c_A$ for all variables x and all constant symbols c, and $\hat{\alpha}(op(t_1, \ldots, t_n)) = op_A(\hat{\alpha}(t_1), \ldots, \hat{\alpha}(t_n))$ for all $op(t_1, \ldots, t_n) \in T_\Sigma(X)$. If t is a variable-free term, then $\hat{\alpha}(t)$ is denoted by t_A.

A *label alphabet* is a pair $\mathcal{C} = (\mathcal{C}_V, \mathcal{C}_E)$, where $\mathcal{C}_V$ is a set of *node labels* and $\mathcal{C}_E$ is a set of *edge labels*. A *partially labelled graph* over $\mathcal{C}$ is a system $G = (V_G, E_G, s_G, t_G, l_{G,V}, l_{G,E})$, where V_G and E_G are finite sets of *nodes* and *edges*, $s_G, t_G\colon E_G \to V_G$ are *source* and *target* functions for edges, $l_{G,V}\colon V_G \to \mathcal{C}_V$ is the partial node labelling function and $l_{G,E}\colon E_G \to \mathcal{C}_E$ is the partial edge

labelling function[1]. A graph is *totally labelled* if $l_{G,V}$ and $l_{G,E}$ are total functions. We write $\mathcal{G}(\mathcal{C})$ for the set of partially labelled graphs, and $\mathcal{G}^t(\mathcal{C})$ for the set of totally labelled graphs over $\mathcal{C}$.

A *premorphism* $g\colon G \to H$ between two graphs G and H consists of two source and target preserving functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$, that is, $s_H \circ g_E = g_V \circ s_G$ and $t_H \circ g_E = g_V \circ t_G$. If g also preserves labels in the sense that $l_H(g(n)) = l_G(n)$ for all n in $\mathrm{Dom}(l_{G,V})$ and $\mathrm{Dom}(l_{G,E})$, then it is a *graph morphism*. Moreover, g is *injective* if g_V and g_E are injective, and it is an *inclusion* if $g(n) = n$ for all nodes and edges n in G.

Assumption 1 We assume a signature $\Sigma = (S, OP)$ such that $\mathtt{Bool} \in S$, $OP_{\lambda,\mathtt{Bool}} = \{\mathtt{true}, \mathtt{false}\}$, $OP_{\mathtt{Bool},\mathtt{Bool}} = \{\neg\}$ and $OP_{\mathtt{BoolBool},\mathtt{Bool}} = \{\wedge, \vee, \rightarrow, \leftrightarrow\}$. The signature is interpreted in a fixed Σ-algebra A such that $A_{\mathtt{Bool}} = \{\mathtt{tt}, \mathtt{ff}\}$, $\mathtt{true}_A = \mathtt{tt}$, $\mathtt{false}_A = \mathtt{ff}$ and $\neg_A, \wedge_A, \vee_A, \rightarrow_A, \leftrightarrow_A$ are the usual Boolean operations. We also assume a family of variables $X = (X_s)_{s \in S}$ and that S contains two distinguished sorts s_V and s_E for nodes and edges. The label alphabets $\mathcal{C}_T$ and $\mathcal{C}_A$ are defined by

$$\mathcal{C}_T = (T_{OP,s_V}(X), T_{OP,s_E}(X)) \quad \text{and} \quad \mathcal{C}_A = (A_{s_V}, A_{s_E}).$$

3 Rules and Rule Schemata

We recall the definition of double-pushout rules with relabelling given in [9], before introducing rule schemata over $\mathcal{G}(\mathcal{C}_T)$.

Definition 1 (Rule). A *rule* $r = (L \leftarrow K \rightarrow R)$ consists of two graph morphisms $K \to L$ and $b\colon K \to R$ over $\mathcal{G}(\mathcal{C}_A)$ such that $K \to L$ is an inclusion and

(1) for all $n \in L$, $l_L(n) = \perp$ implies $n \in K$ and $l_R(b(n)) = \perp$, and
(2) for all $n \in R$, $l_R(n) = \perp$ implies $l_L(n') = \perp$ for exactly one $n' \in b^{-1}(n)$.

The rule r is *injective* if $b\colon K \to R$ is injective. All rules in the graph programs for Dijkstra's algorithm in Section 7 will be injective, but in general we want to allow non-injective rules.

Definition 2 (Direct derivation). Let G and H be graphs in $\mathcal{G}^t(\mathcal{C}_A)$ and $r = (L \leftarrow K \rightarrow R)$ a rule. A *direct derivation* from G to H by r consists of two natural pushouts[2] as in Figure 1, where $L \to G$ is injective.

We write $G \Rightarrow_{r,g} H$ or just $G \Rightarrow_r H$ if there exists a direct derivation as in Definition 2. If $\mathcal{R}$ is a set of rules, then $G \Rightarrow_{\mathcal{R}} H$ means that there is some r in $\mathcal{R}$ such that $G \Rightarrow_r H$. Figure 2 shows an example of a rule where we assume $A_{s_V} = A_{s_E} = \mathbb{R}$. (In pictures like this, numbers next to the nodes are used to represent graph morphisms.)

[1] Given a partial function $f\colon A \to B$, the set $\mathrm{Dom}(f) = \{x \in A \mid f(x) \text{ is defined}\}$ is the *domain* of f. We write $f(x) = \perp$ if $f(x)$ is undefined.

[2] A pushout is *natural* if it is also a pullback. See [9] for the construction of natural pushouts over partially labelled graphs.

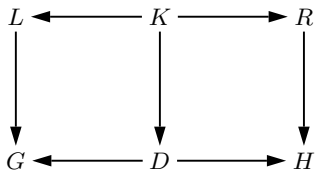


Fig. 1. A direct derivation.

Definition 3 (Match). Given a rule $r = (L \leftarrow K \rightarrow R)$ and a graph G in $\mathcal{G}^t(\mathcal{C}_A)$, an injective graph morphism $g\colon L \rightarrow G$ is a *match* for r if it satisfies the *dangling condition*: no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$.

In [9] it is shown that, given r and an injective morphism $g\colon L \rightarrow G$, there exists a direct derivation as in Figure 1 if and only if g is a match for r. Moreover, in this case D and H are determined uniquely up to isomorphism.

Definition 4 (Rule schema). If $K \rightarrow L$ and $K \rightarrow R$ are graph morphisms over $\mathcal{G}(\mathcal{C}_T)$ satisfying the conditions of Definition 1, then $r = (L \leftarrow K \rightarrow R)$ is a *rule schema*.

An example of a rule schema is shown in Figure 3, where `x`, `y` and `z` are variables of sort `Real`.

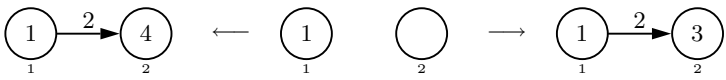


Fig. 2. A rule.

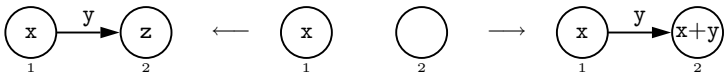


Fig. 3. A rule schema.

Rule schemata are instantiated by evaluating their terms according to some assignment $\alpha\colon X \rightarrow A$.

Definition 5 (Instances of graphs and rule schemata). Given a graph G over $\mathcal{C}_T$ and an assignment $\alpha\colon X \rightarrow A$, the *instance* G^α of G is the graph over $\mathcal{C}_A$ obtained from G by replacing the labelling functions l_G with $\hat{\alpha} \circ l_G$. The instance of a rule schema $r = (L \leftarrow K \rightarrow R)$ is the rule $r^\alpha = (L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha)$.

For example, the rule in Figure 2 is an instance of the rule schema in Figure 3; the associated assignment α satisfies $\alpha(\mathtt{x}) = 1$, $\alpha(\mathtt{y}) = 2$ and $\alpha(\mathtt{z}) = 4$. Note that a rule schema may have infinitely many instances if A contains infinite base sets.

Given graphs G and H in $\mathcal{G}^t(\mathcal{C}_A)$ and a rule schema r, we write $G \Rightarrow_r H$ if there is an assignment α such that $G \Rightarrow_{r^\alpha} H$. For a set $\mathcal{R}$ of rule schemata, $G \Rightarrow_{\mathcal{R}} H$ means that there is some r in $\mathcal{R}$ such that $G \Rightarrow_r H$.

4 Conditional Rules and Conditional Rule Schemata

We introduce *conditional* rule schemata which allow to control the application of a rule schema by comparing values of terms in the left-hand side of the schema. This concept will be crucial to express graph algorithms conveniently.

Analogously to the instantiation of rule schemata to rules, conditional rule schemata will be instantiated to conditional rules. We define a conditional rule as a rule together with a set of admissible matches.

Definition 6 (Conditional rule). A *conditional rule* $q = (r, M)$ consists of a rule $r = (L \leftarrow K \rightarrow R)$ and a set M of graph morphisms such that $M \subseteq \{g\colon L \rightarrow G \mid G \in \mathcal{G}^t(\mathcal{C}_A)$ and g is a match for $r\}$.

Intuitively, M is a predicate on the matches of r in totally labelled graphs. Given a conditional rule $q = (r, M)$ and graphs G and H in $\mathcal{G}^t(\mathcal{C}_A)$, we write $G \Rightarrow_q H$ if there is a morphism g in M such that $G \Rightarrow_{r,g} H$.

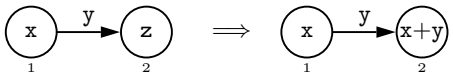
Our concept of a conditional rule is similar to that of [5] where rules are equipped with two sets of morphisms (representing positive and negative application conditions, respectively). Because [5] is based on the so-called single-pushout approach, admissible morphisms need not satisfy the dangling condition.

Conditional rules as defined above are a semantic concept in that the set M of admissible matches will usually be infinite. To represent conditional rules in the syntax of a programming language, we introduce conditional rule schemata which consist of a rule schema and a Boolean term. This term may contain any operation symbols of the predefined signature Σ and, in addition, a special binary predicate `edge` on the nodes of the left-hand side of the rule schema.

Definition 7 (Conditional sule schema). Given a rule schema $(L \leftarrow K \rightarrow R)$, extend the signature Σ to $\Sigma^L = (S^L, OP^L)$ by $S^L = S \cup \{\texttt{Node}\}$, $OP^L_{\lambda,\texttt{Node}} = V_L$, $OP^L_{\texttt{NodeNode},\texttt{Bool}} = \{\texttt{edge}\}$, $OP^L_{w,s} = OP_{w,s}$ if $w \in S^*$ and $s \in S$, and $OP^L_{w,s} = \emptyset$ otherwise. Then a term c in $T_{OP^L,\texttt{Bool}}(X)$ is a *condition* and $\langle (L \leftarrow K \rightarrow R), c\rangle$ is a *conditional rule schema*.

A conditional rule schema is also written as $(L \leftarrow K \rightarrow R)$ `where` c. In pictures, a rule or rule schema $(L \leftarrow K \rightarrow R)$ is often given in the form $L \Rightarrow R$. In this case we assume that K consists of the numbered nodes of L and that these nodes are unlabelled in K. For example, Figure 4 shows a conditional rule schema that is applicable to a graph G only if $\texttt{x}, \texttt{y}$ and $\texttt{z}$ are instantiated such that $\alpha(\texttt{x}) + \alpha(\texttt{y}) < \alpha(\texttt{z})$ and if there is no edge in G from the image of node 2 to the image of node 1.

Conditional rule schemata are instantiated by instantiating the rule schema according to some assignment α and by evaluating the condition by an extension of α which takes into account the meaning of the `edge` predicate.



where $\mathtt{x} + \mathtt{y} < \mathtt{z} \wedge \neg\, \mathtt{edge}(2, 1)$

Fig. 4. A conditional rule schema.

Definition 8 (Instance of a conditional rule schema). Given a conditional rule schema $r = \langle (L \leftarrow K \rightarrow R), c \rangle$, an assignment $\alpha\colon X \to A$ and a graph morphism $g\colon L^\alpha \to G$ with $G \in \mathcal{G}^t(\mathcal{C}_A)$, define the extension $\alpha_g\colon T_{\Sigma^L}(X) \to A$ as follows:

(1) $\alpha_g(x) = \alpha(x)$ and $\alpha_g(c) = c_A$ for all variables x and all constants c in Σ.[3]

(2) $\alpha_g(\mathtt{edge}(v, w)) = \begin{cases} \mathtt{tt} \text{ if there is an edge in } G \text{ from } g(v) \text{ to } g(w), \\ \mathtt{ff} \text{ otherwise.} \end{cases}$

(3) $\alpha_g(op(t_1, \ldots, t_n)) = op_A(\alpha_g(t_1), \ldots, \alpha_g(t_n))$
for all $op(t_1, \ldots, t_n) \in T_{OP^L, S^L}(X)$ with $op \in OP$.

Then the *instance* r^α of r is the conditional rule $\langle (L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha), M \rangle$ where $M = \{g\colon L^\alpha \to G \mid G \in \mathcal{G}^t(\mathcal{C}_A),\ g \text{ is a match and } \alpha_g(c) = \mathtt{tt}\}$.

Given graphs G and H in $\mathcal{G}^t(\mathcal{C}_A)$ and a conditional rule schema $q = r$ where c, we write $G \Rightarrow_q H$ if there is an assignment $\alpha\colon X \to A$ and a graph morphism g such that $G \Rightarrow_{r^\alpha, g} H$ and $\alpha_g(c) = \mathtt{tt}$.

Operationally, the application of a conditional rule schema $(L \leftarrow K \rightarrow R)$ where c to a graph G in $\mathcal{G}^t(\mathcal{C}_A)$ amounts to the following steps:

1. Find an injective premorphism $g\colon L \to G$ satisfying the dangling condition.
2. Find an assignment $\alpha\colon X \to A$ such that for all n in $\mathrm{Dom}(l_L)$, $\hat{\alpha}(l_L(n)) = l_G(g(n))$.
3. Check whether $\alpha_g(c) = \mathtt{tt}$.
4. Construct for $(L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha)$ and g the natural pushouts of Definition 2 (according to [9]).

5 Deterministic Conditional Rule Schemata

For an implementation of a programming language based on rule schemata it is prohibitive to enumerate all instances of a rule schema $r = (L \leftarrow K \rightarrow R)$ in order to find an instance that turns a given premorphism $g\colon L \to G$ into a graph morphism. This is because r may have infinitely many instances. Even if one restricts attention to instances r^α where α evaluates the terms in L to labels of corresponding nodes and edges in G, there may be infinitely many instances left. For example, consider the conditional rule schema in Figure 5 and an associated premorphism $g\colon L \to G$. Whereas the values $\alpha(\mathtt{k})$ and $\alpha(\mathtt{z})$ are uniquely determined by g, there are infinitely many choices for $\alpha(\mathtt{x}), \alpha(\mathtt{y})$ and

[3] Note that α_g is undefined for all constants in $OP^L_{\lambda,\mathtt{Node}}$.

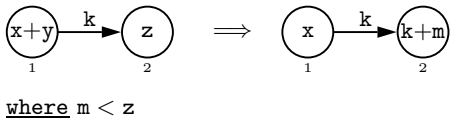


Fig. 5. A conditional rule schema that is not deterministic.

$\alpha(\mathtt{m})$ if nodes are labelled with integers, say. We therefore introduce a subclass of (conditional) rule schemata which are instantiated by premorphisms in at most one way.

A term t in $T_\Sigma(X)$ is *simple* if it is a variable or does not contain any variables. We denote by $\mathrm{Var}(t)$ and $\mathrm{Var}(G)$ the sets of variables occuring in a term t or graph G.

Definition 9 (Deterministic conditional rule schema). A rule schema $(L \leftarrow K \rightarrow R)$ is *deterministic*, if

(1) all labels in L are simple terms, and
(2) $\mathrm{Var}(R) \subseteq \mathrm{Var}(L)$.

A conditional rule schema $\langle r, c\rangle$ with $r = (L \leftarrow K \rightarrow R)$ is deterministic if r is deterministic and $\mathrm{Var}(c) \subseteq \mathrm{Var}(L)$.

For example, the conditional rule schema in Figure 4 is deterministic.

Proposition 1. *Let $r = \langle (L \leftarrow K \rightarrow R), c\rangle$ be a deterministic conditional rule schema and $g\colon L \rightarrow G$ a premorphism with $G \in \mathcal{G}^t(\mathcal{C}_A)$. Then there is at most one instance r' of r such that g is a match for r'.*

Proof. Let r^α and r^β be instances of r such that g is a match for both. By Definition 5 and Definition 8, we have $r^\alpha = r^\beta$ if $\hat{\alpha}(t) = \hat{\beta}(t)$ for all terms t in L and R, and $\alpha_g(c) = \beta_g(c)$ (note that every term in K occurs also in L). Therefore it suffices to show that $\alpha(x) = \beta(x)$ for each variable x in $\mathrm{Var}(L) \cup \mathrm{Var}(R) \cup \mathrm{Var}(c)$. Since r is deterministic, we have $x \in \mathrm{Var}(L)$. Hence there is a node or an edge in L that is labelled with a term containing x. Without loss of generality let v be a node such that $x \in \mathrm{Var}(l_{L,V}(v))$. Because all terms in L are simple, $x = l_{L,V}(v)$. Thus, by Definition 5, $\alpha(x) = \hat{\alpha}(x) = \hat{\alpha}(l_{L,V}(v)) = l_{G,V}(g_V(v)) = \hat{\beta}(l_{L,V}(v)) = \hat{\beta}(x) = \beta(x)$. □

Proposition 1 ensures that premorphisms cannot "instantiate" deterministic (conditional) rule schemata in more than one way. The next proposition gives a necessary and sufficient condition for such an instantiation to take place. The condition makes precise how to find an assignment α as required in the second step of the description of rule-schema application, given at the end of Section 4.

Proposition 2. *Let $g\colon L \rightarrow G$ be a premorphism where $L \in \mathcal{G}(\mathcal{C}_T)$ is labelled with simple terms and $G \in \mathcal{G}^t(\mathcal{C}_A)$. Then there is an assignment $\alpha\colon X \rightarrow A$ such that g is a graph morphism from L^α to G, if and only if for all nodes and edges n, n' in L,*

(1) $l_G(g(n)) = t_A$ *if* $l_L(n)$ *is a variable-free term* t*, and*
(2) $l_G(g(n)) = l_G(g(n'))$ *if* $l_L(n) = l_L(n') \in X$*.*

Proof. Suppose first that g is a graph morphism from L^α to G. If n is labelled with a variable-free term t in L, then n's label in L^α is $\hat{\alpha}(t) = t_A$. Since g is label-preserving, $g(n)$ is labelled with t_A, too. Moreover, if n and n' are labelled with the same variable x in L, then both are labelled with $\alpha(x)$ in L^α. Hence $l_G(g(n)) = \alpha(x) = l_G(g(n'))$.

Conversely, suppose that conditions (1) and (2) are satisfied. For every sort s in S, let d_s be a fixed element in A_s. Then, by (2),

$$\alpha(x) = \begin{cases} l_G(g(n)) & \text{if there is a node or edge } n \text{ with } l_L(n) = x, \\ \mathrm{d}_s & \text{otherwise, where } x \in X_s \end{cases}$$

defines an assignment $\alpha\colon X \to A$. Consider any node or edge n in L^α. If $l_L(n)$ is variable-free, then (1) gives $l_G(g(n)) = t_A = \hat{\alpha}(t) = l_{L^\alpha}(n)$. Otherwise $l_L(n)$ is a variable x, and hence by definition of α, $l_G(g(n)) = \alpha(x) = l_{L^\alpha}(n)$. Thus $g\colon L^\alpha \to G$ is label-preserving. □

6 Graph Programs

We extend the language of [8, 7] by replacing rules with deterministic conditional rule schemata and adding a while-loop.

Definition 10 (Syntax of programs). *Programs* are defined as follows:

(1) For every finite set R of deterministic conditional rule schemata, R and $R{\downarrow}$ are programs.
(2) For every graph B in $\mathcal{G}(\mathcal{C}_T)$ and program P, `while` B `do` P `end` is a program.
(3) If P and Q are programs, then $P; Q$ is a program.

A finite set of conditional rule schemata is called an *elementary* program. Our syntax is ambiguous because a program $P_1; P_2; P_3$ can be parsed as both $(P_1; P_2); P_3$ and $P_1; (P_2; P_3)$. This is irrelevant however as the semantics of sequential composition will be relation composition which is associative.

Next we define a relational semantics for programs. Given a binary relation $\phi \subseteq A \times B$ between two sets A and B, the *domain* of ϕ is the set $\mathrm{Dom}(\phi) = \{a \in A \mid a\,\phi\,b \text{ for some } b \in B\}$. If $A = B$ we write ϕ^* for the reflexive-transitive closure of ϕ. The composition of two relations ϕ and ϱ on A is the relation $\phi \circ \varrho = \{\langle a, c\rangle \mid a\,\phi\,b \text{ and } b\,\varrho\,c \text{ for some } b\}$. Given a graph B in $\mathcal{G}(\mathcal{C}_T)$, let $B? = \{(B \leftarrow B \rightarrow B)\}$ with $B \to B$ being the identity morphism on B.

Definition 11 (Semantics of programs). The *semantics* of a program P is a binary relation $[\![P]\!]$ on $\mathcal{G}^t(\mathcal{C}_A)$[4] which is inductively defined as follows:

[4] Strictly speaking, the graphs in $\mathcal{G}^t(\mathcal{C}_A)$ should be considered as *abstract graphs*, that is, as isomorphism classes of graphs. For simplicity we stick to ordinary graphs and consider them as representatives for isomorphism classes; see [8, 7] for a precise account.

(1) For every elementary program R, $[\![R]\!] = \Rightarrow_R$.
(2) $[\![R{\downarrow}]\!] = \{\langle G, H\rangle \mid G \Rightarrow_R^* H \text{ and } H \notin \mathrm{Dom}(\Rightarrow_R)\}$.
(3) $[\![\underline{\texttt{while}}\ B\ \underline{\texttt{do}}\ P\ \underline{\texttt{end}}]\!] = \{\langle G, H\rangle \in [\![B?; P]\!]^* \mid H \notin \mathrm{Dom}([\![B?]\!])\}$.
(4) $[\![P; Q]\!] = [\![P]\!] \circ [\![Q]\!]$.

By clause (3), the operational interpretation of $\underline{\texttt{while}}\ B\ \underline{\texttt{do}}\ P\ \underline{\texttt{end}}$ is that P is executed as long as B occurs as a subgraph. In particular, the loop has no effect on a graph G not containing B: in this case we have $G\ [\![\underline{\texttt{while}}\ B\ \underline{\texttt{do}}\ P\ \underline{\texttt{end}}]\!]\ H$ if and only if $G = H$. Note also that if G contains B but P fails on input G either because a set of rules in P is not applicable or because P does not terminate, then the whole loop fails in the sense that there is no graph H such that $G\ [\![\underline{\texttt{while}}\ B\ \underline{\texttt{do}}\ P\ \underline{\texttt{end}}]\!]\ H$.

Consider now subsets $\mathcal{G}_1$ and $\mathcal{G}_2$ of $\mathcal{G}^t(\mathcal{C}_A)$ and a relation $\phi \subseteq \mathcal{G}_1 \times \mathcal{G}_2$. We say that a program P *computes* ϕ if $\phi = [\![P]\!] \cap (\mathcal{G}_1 \times \mathcal{G}_2)$, that is, if ϕ coincides with the semantics of P restricted to $\mathcal{G}_1$ and $\mathcal{G}_2$. This includes the case of partial functions $\phi\colon \mathcal{G}_1 \to \mathcal{G}_2$, which are just special relations.

7 Dijkstra's Shortest Path Algorithm

The so-called single-source shortest path algorithm by Dijkstra [1, 11] computes the distances between a given start node and all other nodes in a graph whose edges are labelled with nonnegative numbers. Given a graph G and nodes v and w, a *path* from v to w is a sequence $e_1, \ldots, e_n$ of edges such that $s_G(e_1) = v$, $t_G(e_n) = w$ and $t_G(e_i) = s_G(e_{i+1})$ for $i = 1, \ldots, n-1$. The *distance* of such a path is the sum of its edge labels. A *shortest path* between two nodes is a path of minimal distance.

Dijkstra's algorithm stores the distance from the start node to a node v in a variable $d(v)$. Initially, the start node gets the value 0 and every other node gets the value ∞. Nodes for which the shortest distance has been computed are added to a set S, which is empty in the beginning. In each step of the algorithm, first a node w from $V_G - S$ is added to S, where $d(w)$ is minimal. Then for each edge e outgoing from w, $d(t_G(e))$ is changed to $\min(d(t_G(e)),\ d(w) + l_{G,E}(e))$.

7.1 A Simple Graph Program for Dijkstra's Algorithm

Before giving our graph programs, we specify the signature Σ and the algebra A of Assumption 1. The programs will store calculated distances as node labels, so we need some numerical type for both edge and node labels. Let `Real` be a sort in Σ, $s_V = s_E = \texttt{Real}$, and let $\mathbb{R}^+$ be the set of nonnegative real numbers. We assume the following operation symbols in Σ:[5] $OP_{\lambda,\texttt{Real}} = \mathbb{R}^+ \cup \{\infty, *, \square\}$, $OP_{\texttt{RealReal,Bool}} = \{<\}$ and $OP_{\texttt{RealReal,Real}} = \{+\}$. The algebra A is given by $A_{\texttt{Real}} = \mathbb{R}^+ \cup \{\infty, *, \square\}$, $c_A = c$ for all $c \in OP_{\lambda,\texttt{Real}}$, $x <_A y = \texttt{tt}$ if and only if

[5] Note that all numbers in $\mathbb{R}^+$ are used as constant symbols. The representation of numbers in an implementation of our programming language is beyond the scope of this paper.

$(x, y \in \mathbb{R}^+$ and $x < y)$ or $(x \neq \infty$ and $y = \infty)$, $x +_A y = x + y$ if $x, y \in \mathbb{R}^+$ and $x +_A y = \infty$ otherwise.

Our first program for Dijkstra's algorithm, `Simple_Dijkstra`, is given in Figure 6. We assume that the program is started from a graph in $\mathcal{G}^t(\mathcal{C}_A)$ whose edges are labelled with nonnegative numbers and whose start node is marked by a unique loop labelled with $*$. The rule schema `S_Prepare` relabels every node of the input graph with ∞, `S_Start` deletes the unique loop and relabels the start node with 0, and `S_Reduce` changes a stored distance whenever a shorter path has been found.

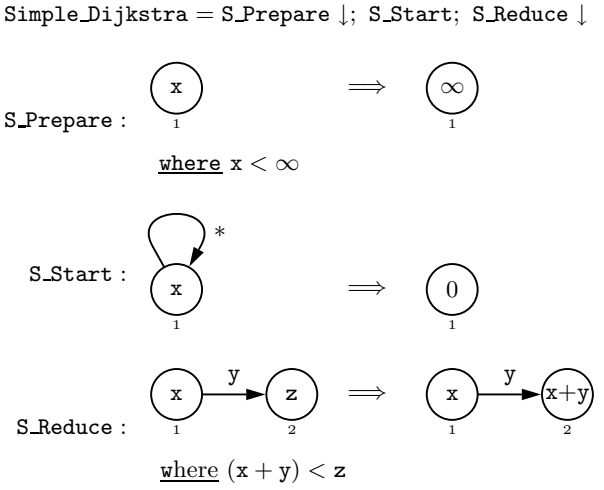


Fig. 6. The program `Simple_Dijkstra`.

Proposition 3 (Correctness of `Simple_Dijkstra`). *Let G be a graph in $\mathcal{G}^t(\mathcal{C}_A)$ containing a unique loop e, where $l_{G,E}(e) = *$ and $l_{G,E}(e') \in \mathbb{R}^+$ for all other edges e'. When started from G,* `Simple_Dijkstra` *terminates and produces a unique graph H which is obtained from G by removing e and labelling each node v with the shortest distance from $s_G(e)$ to v.*

Proof. Termination of `Simple_Dijkstra` follows from the fact that every application of `S_Prepare` reduces the number of nodes not labelled with ∞, and that every application of `S_Reduce` reduces the sum of all node labels in a graph.

Let now H be a graph such that $G \, [\![\texttt{Simple\_Dijkstra}]\!] \, H$. Since there are no rule schemata for adding or deleting nodes, and `S_Start` is the only rule schema that alters G's edges, it is clear that H can be obtained from G by removing the loop e and relabelling the nodes. Thus, H is uniquely determined if each node v is labelled with the shortest distance from $s_G(e)$ to v. To show the latter, we need the following invariance property.

Claim. Let G $[\![$`S_Prepare`$\downarrow$; `S_Start`$]\!]$ $H_0 \Rightarrow^*_{\texttt{S\_Reduce}} H'$. Then for each node v in H', either $l_{H',V}(v) = \infty$ or $l_{H',V}(v)$ is the distance of a path from $s_G(e)$ to v.

Proof. The proposition holds for H_0, because $s_G(e)$ is labelled with 0 and every other node is labelled with ∞. Moreover, it is easy to see that every application of `S_Reduce` preserves the claimed property. □

Suppose now that there is a node v in H such that $l_{H,V}(v)$ is not the shortest distance from $s_G(e)$ to v. We distinguish two cases.

Case 1: $v = s_G(e)$. Since v is labelled with 0 after the application of `S_Start`, and $l_{H,V}(v) \neq 0$, there must be an application of `S_Reduce` that changes $v's$ label to a negative number. But this contradicts the above claim.

Case 2: $v \neq s_G(e)$. By the above claim, there is a path from $s_G(e)$ to v (as otherwise $l_{H,V}(v) \neq \infty$). Let $e_1, \ldots, e_n$ be a shortest path from $s_G(e)$ to v. Let $v_0 = s_G(e)$ and $v_i = t_H(e_i)$ for $i = 1, \ldots, n$. By Case 1, $l_{H,V}(v_0) = 0$. Hence, there is some k, $1 \leq k \leq n$, such that $l_{H,V}(v_k)$ is not the shortest distance from v_0 to v_k and for $i = 0, \ldots, k-1$, $l_{H,V}(v_i)$ is the shortest distance from v_0 to v_i. Now since $e_1, \ldots, e_n$ is a shortest path to v_n it follows that $e_1, \ldots, e_k$ is a shortest path to v_k and that $e_1, \ldots, e_{k-1}$ is a shortest path to v_{k-1}. So the shortest distance from v_0 to v_k is $\sum_{i=1}^{k-1} l_{H,E}(e_i) + l_{H,E}(e_k) = l_{H,V}(v_{k-1}) + l_{H,E}(e_k)$. As this sum is smaller than $l_{H,V}(v_k)$, `S_Reduce` is applicable to e_k. But this contradicts the fact that $H \notin \mathrm{Dom}(\Rightarrow_{\texttt{S\_Reduce}})$. □

The correctness of `Simple_Dijkstra` was easy to show, however the program can be expensive in the number of applications of the rule schema `S_Reduce`. For example, the right-hand derivation sequence in Figure 7 contains 48 applications of `S_Reduce` and represents the worst-case program run for the given input graph of 5 nodes. In contrast, Dijkstra's algorithms (as sketched at the beginning of this section) changes distances only 10 times when applied to the same graph. Although `Simple_Dijkstra` needs only 4 applications of `S_Reduce` in the best case, there is no guarantee that it does not choose the worst case. We therefore refine `Simple_Dijkstra` by modelling more closely the original algorithm.

7.2 A Refined Program

The program `Dijkstra` of Figure 8 uses a `while`-loop to repeatedly select a node of minimal distance and to update the distances of the target nodes of the outgoing edges of that node. Nodes that have not yet been selected are marked by a $\square$-labelled loop. Removing the $*$-labelled loop from a node by `Next` corresponds to adding that node to the set S of the original algorithm. Note that `Dijkstra` is essentially deterministic: `Min` $\downarrow$ always determines a node of minimal distance among all nodes marked with loops, and `Reduce` is applied only to edges outgoing from this node.

The left-hand derivation sequence of Figure 7 is a worst-case run of `Dijkstra`, containing 26 rule-schema applications. Among these are only 10 applications of `Reduce`, which correspond to the 10 distance changes done by the original algorithm. The next proposition establishes the worst-case complexity of `Dijkstra`

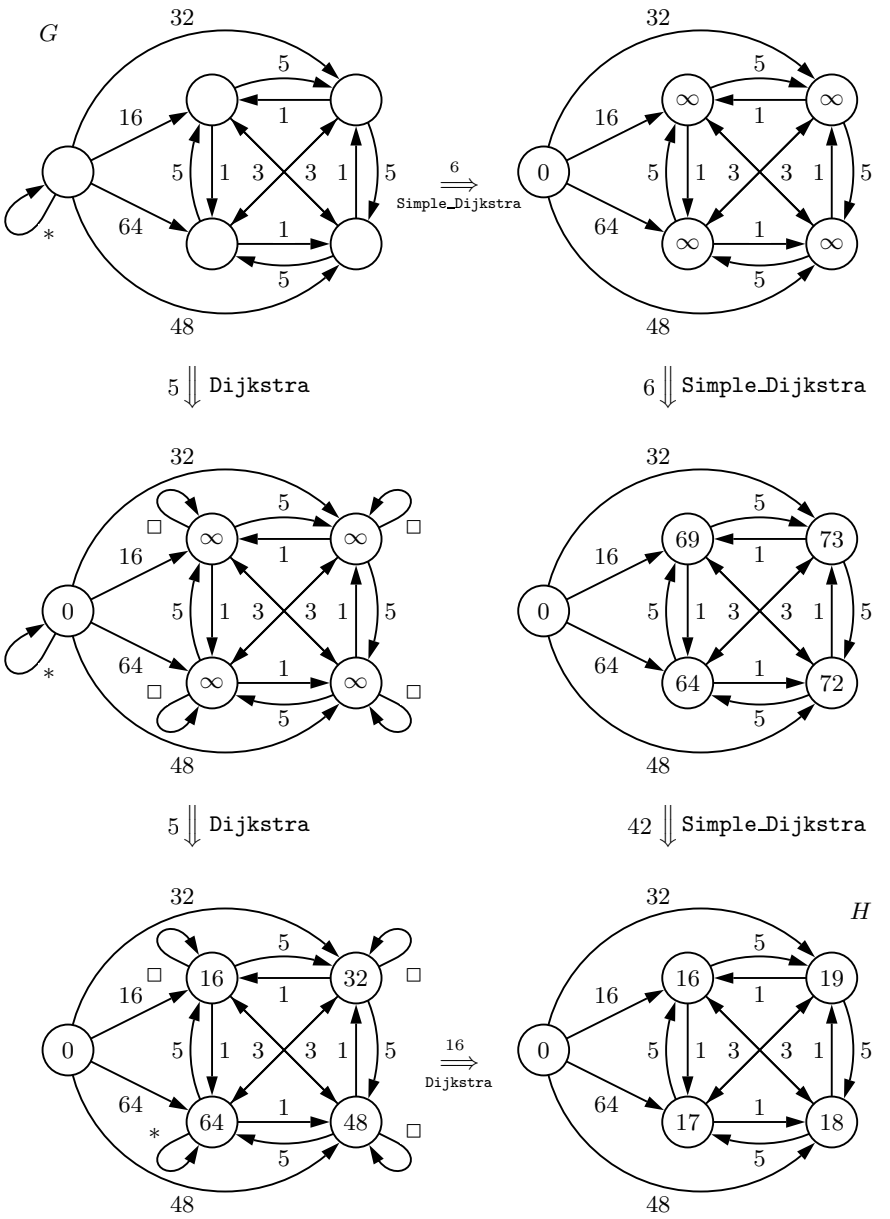


Fig. 7. Derivation sequences of `Simple_Dijkstra` and `Dijkstra`.

in terms of the number of rule-schema applications, where we assume that input graphs satisfy the precondition of Proposition 3.

Proposition 4 (Complexity of `Dijkstra`). *When started from a graph containing n nodes and e edges,* `Dijkstra` *terminates after $O(n^2 + e)$ rule-schema applications.*

Dijkstra = Prepare ↓; Start; while B do Min ↓; Reduce ↓; Next end; CleanUp

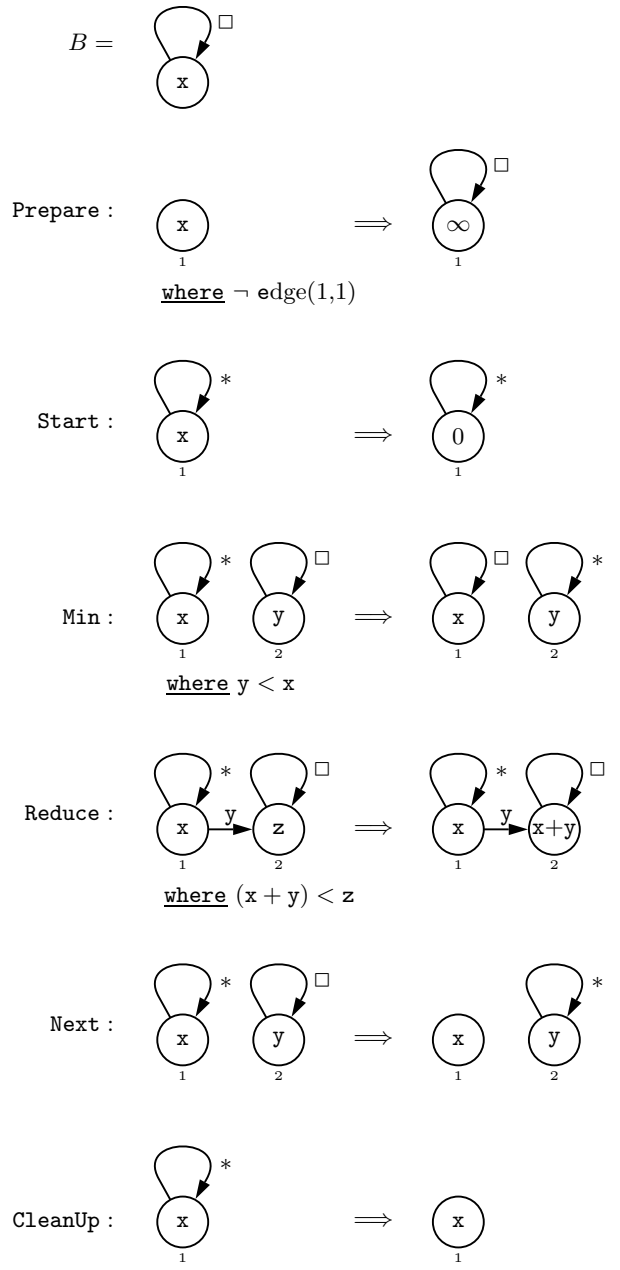


Fig. 8. The program Dijkstra.

Proof. The initialisation phase `Prepare`↓; `Start` uses n rule-schema applications. The body of the **`while`**-loop is executed $(n-1)$-times because initially there are $n-1$ loops labelled with $\square$, and each execution of the body reduces this number by one. So the overall number of `Next`-applications is $n-1$, too. Each execution of `Min`↓ takes at most $n-1$ steps because there is only one $*$-labelled loop. Hence, there are at most $(n-1)^2$ applications of `Min` overall. The total number of `Reduce`-applications is at most e since `Reduce` cannot be applied twice to the same edge. This is because `Reduce` is applied only to edges outgoing from the $*$-marked node, and the $*$ mark is removed by `Next`. Thus, a bound for the overall number of rule-schema applications is $n + (n-1) + (n-1)^2 + e$, which is in $O(n^2 + e)$. $\square$

Note that if we forbid parallel edges in input graphs, then e is bounded by n^2 and hence the complexity of `Dijkstra` is $O(n^2)$.

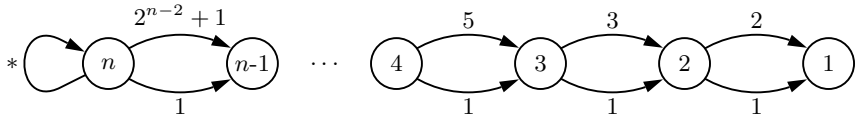


Fig. 9. A worst-case input for `Simple_Dijkstra`.

The quadratic complexity of `Dijkstra` means a drastic improvement on the running time of `Simple_Dijkstra` which may be exponential. More precisely, one can show that for every $n \geq 2$ there is a graph with n nodes and $2(n-1)$ edges such that there is a run of `Simple_Dijkstra` in which the rule schema `S_Reduce` is applied $\sum_{k=1}^{n-1} 2^k$ times. Such a graph is shown in Figure 9. (The running time of `Dijkstra` for this graph is actually linear.)

8 Related Work

A guiding principle in our ongoing design of the graph programming language GP is syntactic and semantic simplicity, which distinguishes GP from the complex PROGRES language [15]. It remains to be seen how much we have to compromise this principle to enable practical programming in application areas. Our approach also differs from a language such as AGG [4] in that we insist on a formal semantics. We want GP to be semantics-based since we consider the ability to formally reason on programs as a key feature.

The rule schemata intoduced in this paper are not the only way to extend graph transformation with calculations on labels. An alternative is to use one of the approaches to attributed graph transformation that have been proposed in the literature. The recent papers [10, 3], for example, merge graphs and algebras so that attributed graphs are usually infinite. We rather prefer to work with finite graphs in which "attributes" are ordinary labels.

Our method of working with rule schemata and their instances is close to Schied's approach to double-pushout transformations on graphs labelled with

algebra elements [14]. (A single-pushout version of this approach is outlined in [13].) Roughly, his double-pushout diagrams can be decomposed into our diagrams with rule schema instantiations on top of them. A major difference between the present paper and [14] is that our rules can relabel and merge items whereas the rules in [14] are label preserving and injective. Schied also introduces conditions for rules, in the form of propositional formulas over term equations, but he does not consider built-in predicates on the graph structure such as our `edge` predicate.

Surprisingly, there seems to be hardly any work on studying graph algorithms in the framework of graph transformation languages. We are only aware of a case study on Floyd's all-pairs shortest path algorithm in Kreowski and Kuske's paper [12]. The paper presents a program for Floyd's algorithm and proves its correctness as well as a cubic bound for the number of rule applications. (The program consists of rules with parameters, similar to our rule schemata, but [12] does not give a general formalism for such rules.)

9 Conclusion

As pointed out in the Introduction, this paper is only the first step in extending the language of [7] to a graph programming language GP. We have introduced graph programs over rule schemata to incorporate numerical data and other basic data types. Rule schemata can have Boolean application conditions which may contain built-in predicates on the graph structure. We have identified deterministic conditional rule schemata as a class of schemata that admit a reasonable implementation in that their applicability and the graphs resulting from applications are uniquely determined by premorphisms from left-hand sides into graphs. As a case study for extended graph programs, we have given two programs for Dijkstra's shortest path algorithm and have analysed their correctness and complexity.

In future work, more case studies on graph algorithms and in other areas will be pursued to find out what additional programming constructs are needed to make GP a practical language. We hope that new constructs can be mapped to a small core of GP – possibly the language used in this paper – to keep the semantics comprehensible and to facilitate formal reasoning on programs, static program analysis, program transformation, etc. And, of course, GP should eventually be implemented so that its practical usefulness can be proved.

References

1. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 2000.
2. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation — Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 3, pages 163–245. World Scientific, 1997.

3. H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004. This volume.
4. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, pages 551–603. World Scientific, 1999.
5. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.
6. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
7. A. Habel and D. Plump. A minimal and complete programming language for graph transformation. In preparation.
8. A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS '01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.
9. A. Habel and D. Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.
10. R. Heckel, J. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 02)*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.
11. D. Jungnickel. *Graphs, Networks and Algorithms*. Springer-Verlag, 2002.
12. H.-J. Kreowski and S. Kuske. Graph transformation units and modules. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 15, pages 607–638. World Scientific, 1999.
13. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, 1993.
14. G. Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und Verteilte Regelsysteme*. Doctoral dissertation, Universität Erlangen-Nürnberg, 1992. Volume 25(2) of *Arbeitsberichte des Instituts für Informatik* (in German).
15. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.